AFRL-AFOSR-UK-TR-2013-0059



# Software Regression Verification

**Professor Ofer Strichman**

**Technion Israel Institute of Technology**
**Information Systems Engineering**
**Technion City, Haifa 32000 Israel**

**EOARD Grant 11-3006**

Report Date: December 2013

Final Report from 15 November 2010 to 14 November 2013



**Air Force Research Laboratory**
**Air Force Office of Scientific Research**
**European Office of Aerospace Research and Development**
**Unit 4515, APO AE 09421-4515**

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From – To)* |
|---|---|---|
| 11 December 2013 | Final Report | 15 November 2010 – 14 November 2013 |

**4. TITLE AND SUBTITLE**

**Software Regression Verification**

**5a. CONTRACT NUMBER**

**FA8655-11-1-3006**

**5b. GRANT NUMBER**

**Grant 11-3006**

**5c. PROGRAM ELEMENT NUMBER**

61102F

**6. AUTHOR(S)**

Professor Ofer Strichman

**5d. PROJECT NUMBER**

**5d. TASK NUMBER**

**5e. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Technion Israel Institute of Technology
Information Systems Engineering
Technion City, Haifa 32000 Israel

**8. PERFORMING ORGANIZATION REPORT NUMBER**

N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

EOARD
Unit 4515
APO AE 09421-4515

**10. SPONSOR/MONITOR'S ACRONYM(S)**

AFRL/AFOSR/IOE (EOARD)

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**AFRL-AFOSR-UK-TR-2013-0059**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

**Distribution A: Approved for public release; distribution is unlimited.**

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

The objective of this research effort was to study and develop a formal methodology for conducting automated or semi-automated software regression testing. To that end, we have developed and extended a regression verification tool (RVT) in various ways, which together improved tremendously its functionality, robustness and completeness. Recall that the main problem that RVT addresses is that of deciding whether two similar programs are equivalent under an arbitrary yet equal context, given some definition of equivalence. More precisely, given a (possibly partial) mapping between the functions of the two programs, the problem is to show which of them are equivalent in an arbitrary context. The basic definition of equivalence that we support is called partial equivalence, which means that two programs emit equal outputs given the same inputs, or at least one of them does not terminate. For most useful definitions of equivalence this problem is undecidable, as is the problem of partial correctness (what most people refer to as general program verification) in general.

**15. SUBJECT TERMS**

EOARD, software engineering, regession verification, formal logic

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18, NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | James H Lawton, PhD |
| UNCLAS | UNCLAS | UNCLAS | SAR | 24 | 19b. TELEPHONE NUMBER *(Include area code)* +44 (0)1895 616187 |

# Final report

**Contract FA8655-11-1-3006**
**Submitted to the European Office of Aerospace Research and Development**

Ofer Strichman

Information Systems Engineering, IE, Technion, Haifa, Israel

During the three years of the project we developed our regression verification tool (RVT) in various ways, which together improved tremendously its functionality, robustness and completeness. Recall that the main problem that RVT addresses is that of deciding whether two similar programs are equivalent under an arbitrary yet equal context, given some definition of equivalence. More precisely, given a (possibly partial) mapping between the functions of the two programs, the problem is to show which of them are equivalent in an arbitrary context. The basic definition of equivalence that we support is called partial equivalence, which means that two programs emit equal outputs given the same inputs, or at least one of them does not terminate. For most useful definitions of equivalence this problem is undecidable, as is the problem of partial correctness (what most people refer to as general program verification) in general. The latter is a term suggested by T. Hoare in 1960's, to denote the problem of checking that a certain condition holds at a particular location assuming the program can get there. It can be cast as a reachability problem: can the program reach a certain part of the code that is guarded by the negation of the checked condition? It is not difficult to reduce this problem to a problem of proving partial equivalence: simply make the program emit output '0' if that part of the code is reached, and '1' otherwise, and compare it to a program that only emits '1'. This proves that equivalence is undecidable as well.

Even in cases that equivalence can be determined in theory (for example when there are no loops and recursive calls), there are tremendous technical problems in performing this task when it comes to an industrial programming language such as C, because of issues such as dynamic memory allocation and the ability of programs to access the memory with arbitrary references. A major part of our research is in fact dedicated to such issues: how to enforce isomorphic heaps at the entrance to the two functions, and how to check that they are isomorphic at the exit point.

Many of the technical details of our progress in the last three years were reported in my previous annual reports, so I will only mention them here by title.

- Support for checking *Mutual termination*. Two programs are said to be mutually terminating if they terminate on exactly the same inputs. RVT's ability to prove mutual termination may expose termination errors introduced by a new version of the code. This work has been reported in the masters' thesis of Dima Elenbogen [Ele13] and in an article [EKS12].

– A theoretical investigation into the possibilities and limitations of proving *partial equivalence of multi-threaded programs*. So far this problem has only been studied for the case of single-threaded deterministic programs. We showed a method for regression verification to establish partial equivalence of multithreaded programs. Specifically, we developed two proof-rules that decompose the regression verification between concurrent programs to that of regression verification between sequential functions, a more tractable problem. This ability to avoid composing threads altogether when discharging premises, in a fully automatic way and for general programs, uniquely distinguishes our proof rules from others used for classical verification of concurrent programs. The results of this work are summarized in [CGS11]. We recently also started considering the effect of synchronization primitives and dynamic thread creation.
– Support for programs with Goto statements. As I reported last year in some detail, RVT now supports translating programs with goto-statements into one with While loops, which can then be handled in the standard flow of RVT.
– We developed several techniques for improving completeness. The development of these methods was driven by our experience with checking real programs. I describe these techniques in detail in Appendix A. They were only published thus far as part of a thesis [Ele13], and we currently work on an extended version of [EKS12] that will include them as well.

We are currently working on new proof rules, based on computing weakest pre-conditions expressions, and applying $k$-induction. Some details about this effort is given in Appendix B.

Finally, we are currently in the midst of improving RVT's capability to handle recursive data structures. Performing regression verification over programs that incorporate pointers, arrays and recursive data structures poses several challenges. When asserting the equality of arrays we must assert the equality of all their members. When checking the equality of structures, we must check equality on all of its members recursively. When checking the equality of pointers, asserting the equality of the addresses is meaningless; we must dereference them and check the actual value. When these pointers point to a recursive data structure, the comparison should be pairwise over all the leaves of the structure. Many of these issues were solved with a tool called UNITRV, built recently by Daniel Kroening based on the infrastructure of CPROVER. This tool can prove the partial equivalence of two functions, even if their inputs include pointers to recursive data structures. It effectively assumes that locations pointed to by an equivalent access path, contain the same data. For example, if both functions access `p -> next -> next -> data`, and `p, next, data` are mapped to one another in the two functions, then they read the same value initially. We are currently working on fully integrating this tool into RVT, in order to improve its completeness, and reach the point in which RVT can verify the equivalence of real industrial programs. So far the issue of pointers was a hurdle in reaching this point.

To conclude, I would like to thank the AF for this generous grant with which I funded this research.

## References

[CGS11] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In *Proc. of 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*, 2011.

[EKS12] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. Proving mutual termination of programs. In Armin Biere, editor, *Haifa Verification Conference (HVC)*, 2012.

[Ele13] Dima Elenbogen. Proving mutual termination of programs. Master's thesis, Technion, Israel Institute of Science, 2013.

[FB88] C.N. Fisher and R.J.L. Blanc. *Crafting a Compiler*. The Benjamin-Cummings Series in Computer Science. Benjamin/Cummings, 1988.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Computer Systems*, 9(3):319–349, 1987.

[God08] Benny Godlin. Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology, 2008.

[GS08] Benny Godlinand Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403 – 439, 2008.

[GS11] Benny Godlin and Ofer Strichman. Regression verification. Technical Report IE/IS-2011-02, Technion, 2011. http://ie.technion.ac.il/tech_reports/1306207119_j.pdf.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Computer Systems*, 12(1):26–61, 1990.

[KKP+81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[Nie85] F. Nielson. Program transformation in a denotational setting. *ACM Trans. Prog. Lang. Sys.*, 7:359–379, 1985.

[NNH05] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag, Berlin, 2005.

[SSS00] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In Hunt and Johnson, editors, *Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[WMM95] R.A. Wilhelm, D.A. Maurer, and D. Maurer. *Compiler Design*. International Computer Science Series. Addison-Wesley, 1995.

## A Methods for improving completeness

No sound method of proving mutual termination can be complete because this problem is undecidable, but we should strive to improve the completeness of our approach. The two major reasons of its incompleteness are related to the overapproximation of the real behavior caused by replacing recursive calls with

uninterpreted functions. Refinement of our uninterpreted functions can solve a few of overapproximation-related issues.

However, there exist other reasons for the incompleteness in our approach. In this appendix we will address a few of them that we have coped with. Some of them are applicable to or refine the output of the decomposition algorithm presented in a technical report [GS11] for verification of partial equivalence. Such improvements are valuable for proving mutual termination too because knowing that some functions are partially-equivalent can be beneficial for establishing their mutual termination.

### A.1 Reducing prototypes of loop-replacing functions

Appendix C of [God08] gives a detailed description how loops are replaced with functions. Local variables that are used inside loops are part of the interface of the replacing function, even if they are written-to before being read. The problem is that these variables are local and, hence, receive a non-deterministic value and thus make the uninterpreted functions representing the loop return different values. The following example demonstrates the issue.

*Example 1.* Consider the pair of C programs listed in Fig. 1[1]. Extracting the

```
int main() {
    int y, x = 1;
    while (x < 10) {
        y = 2 + x;
        x = y + y;
    }
    return x*2;
}
```

```
int main'() {
    int x' = 1, y';
    while (x' <= 9) {
        y' = x' + 2;
        x' = 2 * y';
    }
    return x' << 1;
}
```

**Fig. 1.** Two versions of programs each of which contains a loop with an uninitialized variable $y$ ($y'$) which is written-to before ever being read.

loops into separate recursive functions results in the two programs listed in Fig. 2. When partial equivalence of $\langle main, main' \rangle$ is verified, $\langle {main}^{UF}, {main'}^{UF} \rangle$ are generated as listed in Fig. 3. The values of $y$ and $y'$ in $\langle {main}^{UF}, {main'}^{UF} \rangle$, respectively, are non-deterministic. Consequently, not all the arguments passed into calls UF(Loop_main_while1, &$x$, &$y$) and UF'(Loop_main_while1', &$x'$, &$y'$) are considered equal, because direct pointers are considered equal if they point to equal values. Thus those calls are considered different. As a result, $\langle {main}^{UF}, {main'}^{UF} \rangle$ are not considered call-equivalent. Hence, RVT will fail to prove

[1] Hereafter, the syntax of C is slightly violated, for instance, by ending identifiers of side 1 with '.

```
int Loop_main_while1(int *px,
                     int *py)
{
    if (!(*px < 10)) return 0;
    *py = 2 + *px;
    *px = *py + *py;
    return Loop_main_while1(px, py);
}

int main() {
    int y, x = 1;
    Loop_main_while1(&x, &y);
    return x*2;
}
```

```
int Loop_main_while1'(int *px',
                      int *py')
{
    if (!(*px' <= 9)) return 0;
    *py' = *px' + 2;
    *px' = 2 * *py';
    return Loop_main_while1'(px', py');
}

int main'() {
    int x' = 1, y';
    Loop_main_while1'(&x', &y');
    return x' << 1;
}
```

**Fig. 2.** Two versions of programs from Fig. 1 after elimination of their loops.

```
...

int main^UF() {
    int y, x = 1;
    UF(Loop_main_while1, &x, &y);
    return x*2;
}
```

```
...

int main'^UF() {
    int x' = 1, y';
    UF'(Loop_main_while1', &x', &y');
    return x' << 1;
}
```

**Fig. 3.** Parts of the program generated for proving the mutual termination of functions $main$, $main'$, defined in Fig. 2.

$m\text{-}term(main^{UF}, main'^{UF})$.

□

There is no good reason to include the variables of loop-bodies which satisfy the two following conditions, into the argument list of the functions that replace the loop:

C1. before their values are ever read, some value is assigned into them, and
C2. they are no longer used after the body of the loop.

They may become mere local variables in the replacing functions.

```
int Loop_main_while1(int *px) {
    int y;
    if (!(*px < 10)) return 0;
    y = 2 + *px;
    *px = y + y;
    return Loop_main_while1(px);
}

int main() {
    int y, x = 1;
    Loop_main_while1(&x);
    return x*2;
}
```

```
int Loop_main_while1'(int *px') {
    int y';
    if (!(*px' <= 9)) return 0;
    y' = *px' + 2;
    *px' = 2 * y';
    return Loop_main_while1'(px');
}

int main'() {
    int x' = 1, y';
    Loop_main_while1'(&x');
    return x' << 1;
}
```

**Fig. 4.** Two versions of programs from Fig. 1 after replacement of their loops with functions and reduction of variables $y$ and $y'$ from the argument lists of those replacing functions. See Fig. 2 for a comparison.

*Example 2.* Reconsider the programs given in Fig. 1 and note that variable $y$ ($y'$) in function $main$ ($main'$) is initialized every time before being read in the loop-body. In fact, there is a single execution path in that loop-body. Thus, $y$ ($y'$) satisfies condition C1. Moreover, note that it is not used since the end of the loop-body, i.e., it satisfies C2 too. Hence, $py$ ($py'$) may be reduced from the argument list of the loop-replacing function $Loop\_main\_while1$ ($Loop\_main\_while1'$), and, furthermore, $y$ ($y'$) may become a local variable inside it as listed in Fig. 4. Now $m\text{-}term(main, main')$ can be proven.

□

Here is a description of the procedure we apply for detecting variables which satisfy conditions C1 and C2. Validating C1 amounts to checking that a variable is initialized before being read in every computation path in the loop-body block. If it passed the check, C2 should be validated. The latter validation is done using

*live-variables* analysis [Nie85]. If it establishes that the variable has stopped being a live variable by the end of the loop-body, then C2 holds.

Two simple *intraprocedural static analyses* [Hec77] aid to validate C1 in a checked loop-body block. The first analysis, which we call *Write-To (WT)*, for each node of the control flow graph of that block, finds variables that something is written to them in *all* execution paths leading to the node, including writings in this node itself. The nodes of control flow graphs on which we run our analyses are expressions in C-language. WT is a *flow-sensitive forward* [FB88,WMM95] *must* [NNH05] analysis. Based on its results, the second one, called *Read-Uninitialized (RU)*, finds those variables that may be read before something is written to them, i.e., detects potential reads of uninitialized variables. Those variables of the checked loop which are not listed in the results of RU are written-to before being read in this loop-body, i.e., satisfy C1. RU is a flow-sensitive forward *may* [NNH05] analysis. The both analyses are formally defined in Tables 1 and 2.

| ***kill* function** |
|---|
| $kill(B^\ell) = \emptyset$ |
| ***gen* function** |
| $gen(B^\ell) = def(B)$<br>in all other cases:<br>$gen(B^\ell) = \emptyset$ |
| **Data flow equations $\mathsf{WT}^=$** |
| $\mathsf{WT}_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init(S_\star) \\ \bigcap\{\mathsf{WT}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases}$<br>$\mathsf{WT}_{exit}(\ell) = \big((\mathsf{WT}_{entry}(\ell) \setminus kill(B^\ell)) \cup gen(B^\ell)\big)$, where $B^\ell \in blocks(S_\star)$ |

**Table 1.** Definition of WT analysis. This is an *intraprocedural flow-sensitive forward* ($F = flow(S_\star)$) *must* ($\bigsqcup = \bigcap$) analysis. Let $def(n)$ denote the set of the variables updated in the control flow graph node $n$. See Chapter 2 of [NNH05] for understanding the rest of the notations used here.

The described reduction of variables from the argument lists of loop-replacing functions can be useful for proving partial equivalence too.

### A.2 Mapping functions with different numbers of input arguments

Our normal method for regression verification applies a severe restriction on function pairs mapped in $map_{\mathcal{F}}$: for functions $f \in P$ and $f' \in P'$, $\langle f, f' \rangle \in map_{\mathcal{F}}$ only if $f$ and $f'$ have the same list of formal input parameter types. Our method requires this in order to be able to check call-equiv($f^{UF}$, $f'^{UF}$).

| *kill* **function** |
|---|
| $kill(B^\ell) = \emptyset$ |
| ***gen* function** |
| $gen(B^\ell) = \{v \mid v \in use(B) \wedge v \notin \mathsf{WT}_{entry}(\ell)\}$ |
| **Data flow equations $\mathsf{RU}^=$** |
| $\mathsf{RU}_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init(S_\star) \\ \bigcup\{\mathsf{RU}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases}$ |
| $\mathsf{RU}_{exit}(\ell) = \big((\mathsf{RU}_{entry}(\ell) \setminus kill(B^\ell)) \cup gen(B^\ell)\big)$, where $B^\ell \in blocks(S_\star)$ |

**Table 2.** Definition of $\mathsf{RU}$ analysis. This is an *intraprocedural flow-sensitive forward* ($F = flow(S_\star)$) *may* ($\bigsqcup = \bigcup$) analysis. Let *use(n)* denote the set of the variables which are read in the control flow graph node $n$. See Chapter 2 of [NNH05] for understanding the rest of the notations used here.

However, sometimes it is possible to map functions with different number of input arguments, as we will demonstrate in the following example.

*Example 3.* Consider two versions of a program listed in Fig. 5. Functions $h$

```
int h(int x) {
    if (x <= 0)
        return h(1 - x);
    return x;
}
```

```
int h'(int x', int b') {
    if (b' != 0)
        report'("...");
    if (x' <= 0)
        return h'(1 - x', b');
    return x';
}

void report'(const char *s') {
    ...
}
```

**Fig. 5.** Two versions of a program where functions $h$ and $h'$ have different prototypes.

and $h'$ have different numbers of input arguments. However, argument $b'$ affects neither the guarding conditions over recursive calls of $h'$ nor any value passed into those recursive calls which does affect them. The value in $b'$ has no influence on the future recursive calls of $h'$. Thus we can omit it from input comparisons defined in CallEquiv (see [Ele13]). Namely, we can check call-equivalence between $h$ and the function defined in Fig 6.

□

```
int h'↓{b'}(int x') {
    int b' = nondet();
    if (b' != 0)
        report'("...");
    if (x' <= 0)
        return h'↓{b'}(1 - x');
    return x';
}
```

**Fig. 6.** Function $h'_{\downarrow_{\{b'\}}}$, derived from function $h'$ (see Fig 5) by removal of $b'$ from the parameter list into the body of $h'$.

In Sect. A.3 we will formally present a method for detecting such input arguments as $b'$ in function $h'$ from Ex. 3. We coin input arguments which have no influence on the termination of their function *termination-inert*. But now we will describe what we do with them assuming we have detected them.

We begin with the next definition which we need for this description. Given two functions $f$ and $f'$, a *projection* of the parameter list of $f'$ over the parameter list of $f$ is defined as follows:

**Definition 1 ($\pi_f$).** *Given two functions $f$ and $f'$ such that the parameter list of $f$ is a subset of the parameter list of $f'$, and given a vector $\boldsymbol{in'}$ of actual values passed into $f'$, define $\pi_f(\boldsymbol{in'})$ to denote a reduced version of $\boldsymbol{in'}$ after dropping all the arguments that have no match in the parameter list of $f$.*

Now consider two functions $g$ and $g'$ such that $g'$ has a set of extra parameters $B'$ in comparison with $g$. Further assume we have detected that all the parameters of $B'$ are termination-inert in $g'$. We move them from the parameter list of $g'$ into the body of $g'$, i.e., make them simple local variables in $g'$ initialized with a non-deterministic value. Having updated the parameter list of $g'$, we also need to update all the calls of $g'$ correspondingly in the whole program where $g'$ was defined.

**Definition 2 ($f_{\downarrow_B}$).** *Given function $f$ in program $P$ and a set $B$ of this function's input parameters, define $f_{\downarrow_B}$ to denote the function derived from $f$ by:*

- *moving the elements of $B$ from the parameter list of $f$ into the body of $f$;*
- *initializing them with non-deterministic values;*
- *replacing all the calls to $f$ in $P$ with corresponding calls to $f_{\downarrow_B}$.*

An example of a function derived in this manner is given in Fig. 6. Note that for non-empty $B$, function $f_{\downarrow_B}$ is non-deterministic. We must refine the earlier definitions of termination and mutual termination in order to apply them to non-deterministic functions.

- *Termination.* Let $term(f(\boldsymbol{in}))$ denote the fact that $f(\boldsymbol{in})$ terminates for all its possible computations.

– *Non-termination.* Let $\mathit{non\text{-}term}(f(\boldsymbol{in}))$ denote the fact that there is no possible computation of $f(\boldsymbol{in})$ which terminates. Note that for deterministic functions $\mathit{non\text{-}term}((f(\boldsymbol{in}))) \equiv \neg \mathit{term}(f(\boldsymbol{in}))$. However, the latter is not necessarily true for non-deterministic functions. Therefore, we need to redefine *m-term*.
– *Mutual termination.* If either function $f$ or function $f'$ (or both) is non-deterministic, then their mutual termination is defined as follows:

$$\mathit{m\text{-}term}(f, f') \doteq \forall \boldsymbol{in}.\ \big((\mathit{term}(f(\boldsymbol{in})) \leftrightarrow \mathit{term}(f'(\boldsymbol{in}))) \wedge (\mathit{non\text{-}term}(f(\boldsymbol{in})) \leftrightarrow \mathit{non\text{-}term}(f'(\boldsymbol{in})))\big)\ .$$

Once we have achieved that the prototypes of the discussed functions $g$ and ${g'}_{\downarrow_{B'}}$ match, we can check $\mathit{m\text{-}term}(g, {g'}_{\downarrow_{B'}})$. But as far as the mutual termination of $g$ and $g'$ matters, the definition of *m-term* requires the same parameters in both $g$ and $g'$. We need to address the extra parameters of $g'$ in the following refinement of the definition of mutual termination.

**Definition 3 (Mutual termination of functions with respect to projection of parameter list).** *Two deterministic functions $f$ and $f'$ are* mutually terminating with respect to a subset of inputs *if and only if for any input $\boldsymbol{in}$ of $f$ and any input $\boldsymbol{in'}$ of $f'$, the following holds:*

$$\boldsymbol{in} = \pi_f(\boldsymbol{in'}) \rightarrow (\mathit{term}(f(\boldsymbol{in})) \leftrightarrow \mathit{term}(f'(\boldsymbol{in'})))\ .$$

Let $\mathit{m\text{-}term}_{\pi_f}(f, f')$ denote the fact that $f$ and $f'$ mutually terminate with respect to a subset of inputs. The following inference rule allows to derive a conclusion about $\mathit{m\text{-}term}_{\pi_g}(g, g')$:

$$\frac{\langle f, f' \rangle \in \mathit{map}_{\mathcal{F}} \wedge \mathit{m\text{-}term}(f, {f'}_{\downarrow_{B'}})}{\mathit{m\text{-}term}_{\pi_f}(f, f')}\ (\text{M-TERM-}\pi)\ , \tag{1}$$

where $B'$ is the subset of the input parameters of $f'$ missing in $\pi_f$. A proof of its soundness appears in [Ele13].

Note that the modifications which created ${g'}_{\downarrow_{B'}}$ do not necessarily preserve the semantics of $g'$. Consequently, checking $\mathit{m\text{-}term}(g, {g'}_{\downarrow_{B'}})$ usually involves different uninterpreted functions.

*Example 4.* Reconsider functions $h$ and $h'$ listed in Fig 5. Parameter $b'$ of the prototype of $h'$ is a termination-inert input argument. It can be excluded from the parameter list of $h'$. Function ${h'}_{\downarrow_{\{b'\}}}$, listed in Fig 6, and $h$, defined in Fig 5, have the same prototype. Now RVT can prove $\mathit{m\text{-}term}(h, {h'}_{\downarrow_{\{b'\}}})$ and infer[2] $\mathit{m\text{-}term}_{\pi_h}(h, h')$.

[2] In practice, RVT does not make the minute formal distinction between *m-term* and $\mathit{m\text{-}term}_{\pi_f}$ in its output. They are reported in the same manner.

### A.3 Detecting termination-inert input arguments.

An algorithm for checking whether a given argument $v$ of function $f$ is a termination-inert input argument of $f$ consists of two stages. First, it builds a *System Definition Graph* [HRB90] (SDG) for program $P$ where $f$ is defined.

Briefly, an SDG is an extension of a *Program Dependence Graph* [FOW87,KKP+81] (PDG) for multi-function programs. The original nodes of some function's PDG represent the statements of the function. The edges of the PDG represent data and control dependencies between the statements of the function and thus define their partial order: the semantics of the function is preserved if its statements are executed in this order.

An SDG consists of PDGs for each function of the program plus the following additions. Each function $g$ of the program is associated with an entrance node "*Enter* $g$". For each input argument $u$ of this function, the SDG contains a node of type $u = u_{in_g}$ and an edge entering this node and leaving node "*Enter* $g$". Each node representing a call to function $g$ has a leaving edge entering the entrance node of $g$, i.e., "*Enter* $g$". In addition, for an expression *expr* passed as parameter $u$ in that call, there are a node of type $u_{in_g} = expr$ with the two following edges:

- an entering edge which leaves that function-call node, and
- a leaving edge which enters the recently mentioned node of type $u = u_{in_g}$.

The return value of $g$ has its own dedicated node $retval_g$. Its entering edges leave nodes whose statements affect the return value. Its leaving edges enter nodes whose statements depend on the return value. Fig. 7 demonstrates an example of an SDG built for the sub-program starting in function $h'$ from Fig 5.

At the second stage the algorithm checks whether any of the calls to function $f$ is reachable from node $v = v_{in_f}$, where, recall, $v$ is the name of the given input argument. If none is reachable, then argument $v$ is a termination-inert input argument of $f$. The algorithm is presented in Alg. 1.

*Example 5.* Regard the SDG in Fig. 7, built for the sub-program starting in function $h'$ from Fig 5. It has no node of a function call to $h'$ which is reachable from node $b' = b'_{in_{h'}}$. Hence, $b'$ does not affect any guarding condition over any recursive call to $h'$.

**Algorithm 1** Algorithm for checking whether an input argument is termination-inert.

```
1: function IsCallEquivInert(Program P, function f, argument v)
2:     Build an SDG for P;
3:     for each call to f in this SDG do
4:         if this call is reachable from node v = v_{in_f} then return FALSE;
5:     return TRUE;
```

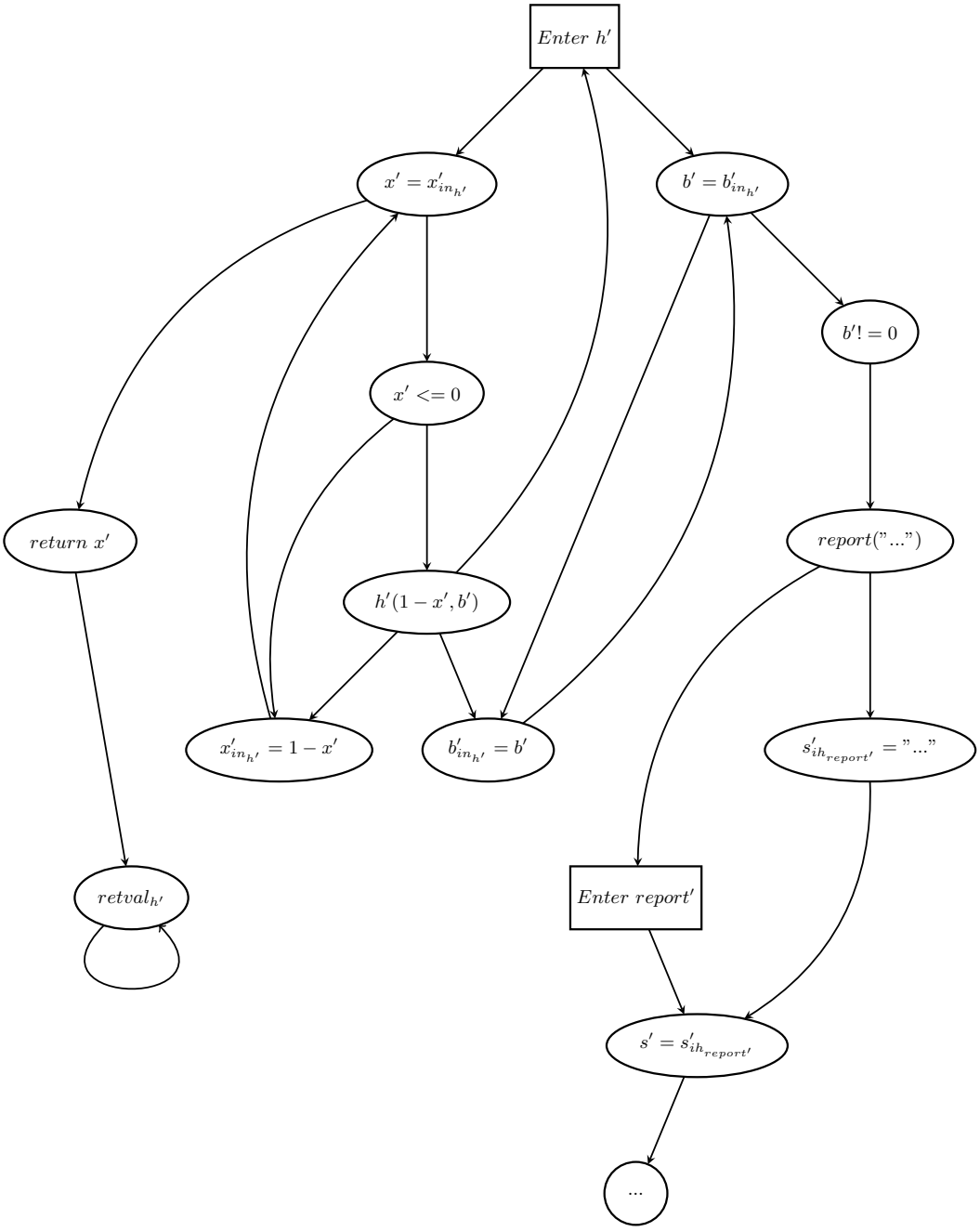


**Fig. 7.** The System Definition Graph [HRB90] of the sub-program starting in function $h'$, defined in Fig 5.

### A.4 Partial equivalence with respect to a subset of outputs

The improvement reported in this section refines the output of the decomposition algorithm for checking partial equivalence [GS11].

Recall that C functions may have multiple outputs, and that so far we defined partial equivalence with respect to all of them, i.e., given the same inputs, the two functions are equivalent in all output elements pair-wise. However, sometimes the equivalence of *some* of the outputs is sufficient for proving mutual termination.

*Example 6.* Consider the functions listed at the top of Fig. 8. Formally, $g$ and $g'$ are not partially equivalent because different values are assigned into $p$ and $p'$, which are among the outputs of $g$ and $g'$, respectively. But the return values of $g$ and $g'$ are equivalent. This fact could be useful for establishing $m\text{-}term(g, g')$.

```
int g(int x, int *p) {
    if (x < 5 || p == NULL)
        return 0;
    *p = 0;
    g(g(x - 1, p), NULL);
    return 0;
}
```

```
int g'(int x', int *p') {
    if (x' < 5 || p' == NULL)
        return 0;
    *p' = 1;
    g'(g'(x' - 1, p'), NULL);
    return 0;
}
```

```
int g^UF(int x, int *p) {
    if (x < 5 || p == NULL)
        return 0;
    *p = 0;
    UF_g(UF_g(x - 1, p), NULL);
    return 0;
}
```

```
int g'^UF(int x', int *p') {
    if (x' < 5 || p' == NULL)
        return 0;
    *p' = 1;
    UF'_g'(UF'_g'(x' - 1, p'), NULL);
    return 0;
}
```

**Fig. 8.** Two versions of functions which are partially equivalent with respect to their return values (at the top) and their isolated versions (at the bottom).

Consider $g^{UF}$ and $g'^{UF}$ listed at the bottom of Fig. 8. The obstacle for proving $call\text{-}equiv(g^{UF}, g'^{UF})$ is the fact that given the same inputs, $UF_g$ and $UF'_{g'}$ are not enforced to produce the same return values. But we may enforce the equivalence of the return values of $UF_g$ and $UF'_{g'}$ only, because $g$ and $g'$ are partially equivalent with respect to their return values.

□

Let *out(f)* denote the list of output elements that function $f$ produces.

**Definition 4 (Partial equivalence of functions with respect to paired elements of the outputs).** *Two functions $f$ and $f'$ are* partially equivalent with respect to $\langle o, o' \rangle$ *such that $o \in out(f) \wedge\ o' \in out(f')$ if any two terminating*

*executions of $f$ and $f'$ starting from the same inputs, produce the same values for $o$ and $o'$.*

Let $p\text{-}equiv_{\langle o,o'\rangle}(f,f')$ denote the fact that $f$ and $f'$ are partially equivalent with respect to $\langle o,o'\rangle$. For given $o \in out(f)$ and $o' \in out(f')$, let $f \overset{o,o'}{=\!=\!=} f'$ denote the fact that given the same inputs $f$ and $f'$ produce the same values for $o$ and $o'$, respectively.

When RVT is activated for checking partial equivalence, it first attempts to establish $p\text{-}equiv(f,f')$ for each $\langle f,\ f'\rangle$ which it is checking. Only if it fails to have proven this, it checks the equivalence of output elements one by one. For each pair of output elements $\langle o,\ o'\rangle$ with respect to which partial equivalence could be proven, it assigns label $part\_eq_{\langle o,o'\rangle}$ to $\langle f,\ f'\rangle$. Thereby it finds a maximal mapping $\{\langle o,o'\rangle \mid o \in out(f) \wedge o' \in out(f')\ \wedge\ p\text{-}equiv_{\langle o,o'\rangle}(f,f')\}$. This mapping can be also useful when the outputs of $f$ cannot be bijectively mapped with the outputs of $f'$.

Now we can refine (enforce-1) (see ([Ele13])):

$$UF_f \overset{o,o'}{=\!=\!=} UF'_{f'} \Rightarrow \big(\langle f,f'\rangle \in map_{\mathcal{F}} \wedge p\text{-}equiv_{\langle o,o'\rangle}(f,f')\big) \text{ (enforce-2)} \qquad (2)$$

We refine the implementation of UF' in a manner compatible with this condition, i.e., given the same inputs, the values of $o$ and $o'$ are the same when $\langle f,\ f'\rangle$ is labeled $part\_eq_{\langle o,o'\rangle}$. Otherwise, the values of the output elements need not be the same. The refined implementation of UF' is shown at the bottom of Fig. 9.

# B New proof methods

In this appendix, we list several research direction that we currently investigate, all of which are targeted towards finding ways to prove equivalence in cases that our current methods fails. Hence, it is also targeted towards completeness. The methods are based on computing weakest-preconditions and $k$-induction.

## B.1 Using Weakest Pre-Condition information

Before we overview the calculation of a WP, we shall recall the definition of a Hoare Triplet $\{Q\}S\{R\}$. This notion states that when a pre-condition $Q$ holds before the execution of $S$ then after the execution of $S$ the predicate $R$ will be evaluated to *true*. A program statement can be viewed as a state transformer which transforms a pre-condition to a post-condition. We calculate a pre-condition by transforming the post-condition each statement at a time backward starting from the termination point. We shall now overview some of the possible statements and how to transform the predicates. The following two lines of code illustrate the calculation:

```
i=z+4;

z=i+z * 2;
```

```
1: function UF(function index g, input parameters in)                    ▷ Called in side 0
2:     if in ∈ params[g] then return the output of the earlier call UF(g, in);
3:     params[g] := params[g] ⋃ in;
4:     return a non-deterministic output;

5: function UF'(function index g', input parameters in')                 ▷ Called in side 1
6:     if in' ∈ params[g'] then return the output of the earlier call UF'(g', in');
7:     params[g'] := params[g'] ⋃ in';
8:     if in' ∈ params[g] then                                           ▷ ⟨g, g'⟩ ∈ map_F
9:         result := [];
10:        for each o_i ∈ out(g) do                                      ▷ o'_i ∈ out(g')
11:            if ⟨g, g'⟩ is marked as part_eq or as part_eq_⟨o_i, o'_i⟩ then
12:                append the result for o_i of the earlier call UF(g, in') to result;
13:            else append a non-deterministic value to result;
14:        return result;
15:    assert(0);                                    ▷ Not call-equivalent: params[g'] ⊈ params[g]
```

**Fig. 9.** Implementations for functions UF and UF', where the latter takes into consideration partial information about partial equivalence. UF and UF' emulate uninterpreted functions if instantiated with functions that are mapped to one another, and form a part of the generated program $\delta$, as shown in CallEquiv (see [Ele13]) or in the determinization thereof. These functions also contain code for recording the parameters with which they are called.

Our desired post-condition is $z > i + 3$. We sequentially calculate the WP. First we calculate the pre-condition of the second statement:

$$WP\left(z = i + z * 2, z > i + 3\right) = z > \frac{3}{2}$$

. Finally we will calculate the pre-condition of the first statement:

$$WP\left(i = z + 4, z > \frac{3}{2}\right) = i > \frac{11}{2}$$

We received our pre-condition. Let us examine another example which includes a conditional statement:

```
if (res % 2 == 0)
    res = res - 1;
else
    res = res + 1;
```

Our desired post-condition is $res > 0$. We receive:

$$WP\left(S, res > 0\right) \equiv \left((res\%2 == 0) \Rightarrow WP\left(res = res - 1, res > 0\right)\right) \wedge$$

$$\left(\neg\left(res\%2 == 0\right) \Rightarrow WP\left(res = res + 1, res > 0\right)\right) \equiv \left((res\%2 == 0) \Rightarrow res > 1\right) \wedge$$

$$\left(\neg\left(res\%2 == 0\right) \Rightarrow res > -1\right)$$

**Motivating example** Consider the following two programs:

```
int f0(int x){
    if (x <= 0){
        return 1;
    }
    int res =  f0(x-1);
    if (res < 0) return 3;
    else return 2;
}
```

```
int f1(int x){
    if (x <= 0){
        return 1;
    }
    int res =  f1(x-1);
    if (res < 0) return 1;
    else return 2;
}
```

**Fig. 10.** The two compared functions.

Both $f0$ and $f1$ are equivalent. One might claim that in case the variable $res$, which is the result of the recursive call, might return a negative value and cause the result value of the two functions to be different. However a closer examination would reveal that both functions never return a negative result and so the expression $res < 0$ always evaluates to $false$. Now we would like to prove

```
int f0(int x){
    if (x <= 0){
        return 1;
    }
    int res =  UF_f0(x-1);
    if (res < 0) return 3;
    else return 2;
}
```

```
int f1(int x){
    if (x <= 0){
        return 1;
    }
    int res =  UF_f1(x-1);
    if (res < 0) return 1;
    else return 2;
}
```

**Fig. 11.** The two functions of Fig. 10, after the function calls are replaced with calls to the same UF.

the equivalence of the two functions using uninterpreted function abstraction. We do this by replacing the recursive calls of $f0$ and $f1$ to their matching uninterpreted functions UF_f0 and UF_f1.

Running CBMC to perform the equivalence proof requires the following *main* function:

```
int main(void){
    int in, out0, out1;
    out0 = f0(in);
    out1 = f1(in);
    assert(out0 == out1);
    return 0;
}
```

**Fig. 12.** The main function calls $f0$ and $f1$ with a nondeterministic input, and compares their return values.

CBMC will output a failure when asserting the claim above. This example manifests the incompleteness of this method. Abstracting away our function calls to uninterpreted functions has resulted in information loss: the UF's do not keep the positive return value property. Whenever an assertion fails CBMC generates a counterexample. In this case, examining the counterexample will demonstrate the root cause of the failure: the UF's had returned negative values. In the next section I shall propose two methods to preserve information in some cases while abstracting recursive calls to UF's.

**Proposed Solutions** In this section I will describe two approaches which can assist in enlarging the set of partially equivalent program pairs that we can prove and thus improve completeness.

**Strengthening UFs using weakest pre-condition** In Figs.10 and 12 we encountered two partially equivalent programs. However our original method

failed to prove their equivalence. Now we shall attempt to solve this problem using WP predicates. The assertion in Fig. 12 is the desired post condition of partial equivalence. By calculating the weakest pre-condition as described in section B.1 we can conclude that the weakest pre-condition required from the the result of the UFs in Fig. 11 is $res_{f0} \geq 0 \wedge res_{f1} \geq 0$

We shall formulate the following proof rule:

$$\frac{\text{p-equiv}(call\ f, call\ f') \wedge WP \vdash_H \text{p-equiv}(call\ f, call\ f') \wedge WP}{\text{p-equiv}(call\ f, call\ f') \wedge WP}$$

The proof rule is very similar to the proof rule at figure **??** with the addition of the conjunction with the WP formula. This conjunction is used to strengthen our induction. Let us return to our example and show how the new strengthened proof rule is used when translated to code. First we remind that uninterpreted functions keep the premise of the partial equivalence rule as shown in [GS08]. Now we add to our code the WP premise. We add the required weakest pre-condition assumption to both our functions. By adding the weakest pre-condition we receive the following code:

```
int f0(int x){
    if (x <= 0){
        return 1;
    }
    int res = UF_f0(x-1);
    assume(res >= 0);
    if (res < 0) return 3;
    else return 2;
}
```

```
int f1(int x){
    if (x <= 0){
        return 1;
    }
    int res = UF_f1(x-1);
    assume(res >= 0);
    if (res < 0) return 1;
    else return 2;
}
```

**Fig. 13.** The two functions of Fig. 11, after adding the weakest pre-condition predicate as an assumption

To have a consistent inductive proof rule we must also alter the main function in the following way:

```
int main(){
    int n, res0, res1;
    ret0 = f0(n);
    ret1 = f1(n);
    assert(ret0 == ret1 && ret0 > 0 && ret1 > 0);
    return 0;
}
```

CBMC returns 'verified' on these two functions, hence we were able to prove the step of the induction. If we examine the process of calculating the WP formula more closely we can see that we start with a premise that both function output the same value and calculate the required condition that must hold at

a certain point in the code so that the premise will be true. From this we can conclude the following relation: $out == out' \iff WP$. It is obvious that we can remove the equality assertion and still receive the same result.

### B.2 *K*-Induction

The standard induction proof rule over the natural numbers can be formulated as follows:

$$P(0) \to (\forall n P(n-1) \to P(n) \to \forall n P(n)) \;, \tag{3}$$

where $P$ is the formula we wish to prove.

A generalization called $k$-induction, was proposed by [SSS00]:

$$\bigwedge_{i=0}^{k-1} P(i) \wedge \forall n \left( \left( \bigwedge_{i=0}^{k-1} P(n+i) \right) \to P(n+k) \right) \to \forall n P(n) \;. \tag{4}$$

Note that this formula is a generalization of (3) when $k = 1$. We propose to use $k$-induction in order to improve the completeness of regression verification.

**Implementing *K*-Induction** Recall the two functions in Fig. 10, and their abstraction in Fig. 11. Recall that the abstract version was *too* abstract, which prevented us from proving partial equivalence. Let us attempt to solve this problem using 2-Induction.

```
int f0_2(int x){
    if (x <= 0){
        return 0;
    }
    int res;
    res = UF_f0(x-1);
    if (res < 0) return 3;
    else return 2;
}

int f0_1(int x){
    if (x <= 0){
        return 0;
    }
    int res = f0_2(x-1);
    record_out_f0_2 = res;
    if (res < 0) return 3;
    else return 2;
}
```

```
int f1_2(int x){
    if (x <= 0){
        return 0;
    }
    int res;
    res = UF_f1(x-1);
    if (res < 0) return 1;
    else return 2;
}

int f1_1(int x){
    if (x <= 0){
        return 0;
    }
    int res = f1_2(x-1);
    assume(res == record_out_f0_2);
    if (res < 0) return 1;
    else return 2;
}
```

**Fig. 14.** The programs of Fig. 11, unrolled twice.

In Fig. 14 we have unrolled the two functions twice. We say that $f0\_1$ and $f1\_1$ are a *first level functions* and $f0\_2$ and $f1\_2$ are a *second level functions*. The functions on the first level call their matching functions on the second level and the second level functions call the matching UFs.

In section **??** we mentioned that when proving the inductive step, the inductive assumption is implemented by the UFs, however to prove the induction step with 2-induction we will need another level of assumption. We implemented this using another assumption in both $f0\_1$ and $f1\_1$ which assumes the equality of the results in the first level. We implemented this using the global variable $record\_out\_f0\_2$ which records the result of side 0 in the second level.

### B.3 Combining $K$-Induction and weakest pre-condition

Combining both methods may improve completeness even further. We implement this by replacing the equivalence assumption with the weakest pre-condition predicate. Fig. 15 exhibits this. Note that the UFs no longer supply us with the required inductive step assumption and therefor we must add an additional assumption.

```
int f0_2(int x){
    if (x <= 0){
        return 0;
    }
    int res;
    res = UF_f0(x-1);
    assume(res > 0);
    if (res < 0) return 3;
    else return 2;
}

int f0_1(int x){
    if (x <= 0){
        return 0;
    }
    int res = f0_2(x-1);
    assume(res > 0);
    record_out_f0_2 = res;
    if (res < 0) return 3;
    else return 2;
}
```

```
int f1_2(int x){
    if (x <= 0){
        return 0;
    }
    int res;
    res = UF_f1(x-1);
    assume(res > 0);
    if (res < 0) return 1;
    else return 2;
}

int f1_1(int x){
    if (x <= 0){
        return 0;
    }
    int res = f1_2(x-1);
    assume(res > 0);
    if (res < 0) return 1;
    else return 2;
}
```

**Fig. 15.** The programs of Fig. 14 with weakest pre-condition assumptions.

When combining the two methods we must separate the base case proof from the inductive step proof. Our original method will succeed in proving the partial equivalence of the two functions in Fig. 16 even though they are not equivalent

due to the negative return value. This proof doesn't fail as it should because the weakest pre-condition assumption blocks all paths where the UFs can return a negative value.

We prove our base case step by making two transformations to our original method. First we convert all UF calls back to recursive calls. We do this by adding a new global boolean variable *base*, when verifying the base case its value is *true* and $false$ otherwise. The mechanism is displayed in the following code:

```
if (base) res = f0_1(x-1);
else res = UF_f0(x-1);
```

Second when the *base* variable is set to *true* we limit CBMC to unwind the recursion to $k$ and run it. Now CBMC will explore all $k$ level base cases. In Fig. 17 we can see the functions from Fig. 16 modified to handle the base case proof. CBMC indeed returns 'un-verified' when given the modified functions.

```
int f0(int x){
    if (x <= 0){
        return 1;
    }
    int res = UF_f0(x-1);
    assume(res >= 0);
    if (res < 0) return 3;
    else return -1;
}
```

```
int f1(int x){
    if (x <= 0){
        return 1;
    }
    int res = UF_f1(x-1);
    assume(res >= 0);
    if (res < 0) return 1;
    else return -1;
}
```

**Fig. 16.** The two functions of Fig. 11 with a negative return value.

```
int f0(int x){
    if (x <= 0){
        return 1;
    }
    if (base) res = f0_2(x-1);
    else res = UF_f0(x-1);
    if (!base) assume(res >= 0);
    if (res < 0) return 3;
    else return -1;
}
```

```
int f1(int x){
    if (x <= 0){
        return 1;
    }
    if (base) res = f1(x-1);
    else res = UF_f1(x-1);
    if (!base) assume(res >= 0);
    if (res < 0) return 1;
    else return -1;
}
```

**Fig. 17.** The two functions of Fig. 16 with base case handling code.

### B.4 Comparing $K$-Induction and weakest pre-condition

In my thesis I would like to explore the relation between the WP method and the $k$-induction partial equivalence proof method. In the example shown before their strength is equal. While 1-induction was unable to prove what WP managed to prove, 2-induction was sufficient. Another interesting topic is the relation between different characteristics of the program and the required level of induction: can this level be computed? can we compute a bound on this level?

### B.5 $K_1, K_2$ Induction

```
int f1(int x){
    n1++;
    if (x <= 1) return 0;
    int res = f1(x - 1) + 1;
    return res;
}
```

```
int f2(int x){
    n2++;
    if (x <= 1) return 0;
    x--;
    if (x <= 1) return 1;
    int res = f2(x - 1) + 2;
    return res;
}
```

**Fig. 18.** Two equivalent recursive programs that iterate a different number of times given the same input.

Consider the example in Fig. 18. It shows an optimization called Loop Unwinding, in a recursive function format. This optimization may result in several benefits such as reduced branch prediction penalties, better parallelization, and more. These optimizations can be produced automatically by the compiler or manually. Often we would like to check partial equivalence between two similar functions where one is some sort of an unwinding optimization of the other.

Our previous method of $k$-induction will fail to prove the equivalence of the two previous functions because the bodies of the functions are not equivalent and also the recursive calls on each side are different. However if we can receive the relation between the unrolling of the functions from the optimizer we can use it to perform a $K_1 - K_2$ induction.